Parallel Computing 40 (2014) 738–753

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Towards unified secure on- and off-line analytics at scale

P. Coetzee *, M. Leeke, S. Jarvis

Department of Computer Science, University of Warwick, United Kingdom

ARTICLE INFO

Article history:
Available online 24 July 2014

Keywords:
Data science
Analytics
Streaming analysis
Hadoop
Domain specific languages
Data intensive computing

ABSTRACT

Data scientists have applied various analytic models and techniques to address the oft-cited problems of large volume, high velocity data rates and diversity in semantics. Such approaches have traditionally employed analytic techniques in a streaming or batch processing paradigm. This paper presents CRUCIBLE, a first-in-class framework for the analysis of large-scale datasets that exploits both streaming and batch paradigms in a unified manner. The CRUCIBLE framework includes a domain specific language for describing analyses as a set of communicating sequential processes, a common runtime model for analytic execution in multiple streamed and batch environments, and an approach to automating the management of cell-level security labelling that is applied uniformly across runtimes. This paper shows the applicability of CRUCIBLE to a variety of state-of-the-art analytic environments, and compares a range of runtime models for their scalability and performance against a series of native implementations. The work demonstrates the significant impact of runtime model selection, including improvements of between 2.3× and 480× between runtime models, with an average performance gap of just 14× between CRUCIBLE and a suite of equivalent native implementations.

1. Introduction

To derive insight and provide value to organisations, data scientists must make sense of a greater volume and variety of data than ever before. In recent years this challenge has motivated significant advances in data analytics, ranging from streaming analysis engines such as IBM's InfoSphere Streams [1], Backtype's Storm [2] or Yahoo!'s S4 [3], to an ecosystem of products built on the MapReduce [4] framework.

When data specialists set out to perform analyses they are typically faced with a decision: they can opt to receive continuous insight but limit analytic capabilities to a functional or agent-oriented streaming architecture, or make use of a bulk data paradigm but risk batch analyses taking hours or even days to complete. It is, of course, possible to maintain systems that target streamed and batch paradigms separately, though this is less desirable than having a single system with the semantics to account for those paradigms in a unified manner. The need to support multiple methodologies presents a further challenge; ensuring analyses are correct and equivalent across platforms. These issues are further complicated by deployment scenarios involving multi-tenant cloud systems or environments with complex access control requirements.

Our research seeks to alleviate many of the issues highlighted above through the development of CRUCIBLE, a framework consisting of (i) a domain specific language (DSL) for describing analyses as a set of communicating sequential processes, (ii) a common runtime model for analytic execution in multiple streamed and batch environments, and (iii) an approach which automates the management of cell-level security labelling uniformly across runtimes. In particular, this paper demonstrates

* Corresponding author.
E-mail addresses: p.l.coetzee@warwick.ac.uk (P. Coetzee), m.leeke@warwick.ac.uk (M. Leeke), s.a.jarvis@warwick.ac.uk (S. Jarvis).

http://dx.doi.org/10.1016/j.parco.2014.07.004
0167-8191/

how CRUCIBLE (named after the containers used in chemistry for high-energy reactions) can be used across multiple data sources to perform highly parallel distributed analyses of data simultaneously in streaming and batch paradigms, efficiently delivering integrated results whilst making best use of existing cloud infrastructure.

The specific contributions of this paper are as follows:

- A high-level DSL and suite of runtime environments, adhering to a common runtime model, that provide consistent execution semantics across on- and off-line data. This is the first DSL designed specifically to target the execution of on- and off-line analytics with equal precedence.
- The development of a new primitive in the developed DSL that permits a single analytic to be run equivalently over multiple data sources: locally, over Accumulo data, and over files in the Hadoop Distributed File System (HDFS).
- A novel framework for the semi-automated management of cell-level security, applied consistently across runtime environments, enabling the management of data visibility in on- and off-line analysis.
- An evaluation of the performance of CRUCIBLE on a set of best-in-class runtime environments, demonstrating framework optimisations that result in an average performance gap of just 14× when compared to a suite of native implementations.

The remainder of this paper is structured as follows: Section 2 provides a summary of related work. Section 3 introduces the CRUCIBLE system and describes its abstract execution model. Section 4 presents a performance analysis and discussion of the CRUCIBLE runtimes and associated optimisations. Finally, Sections 5 and 6 provide avenues for further research and conclude the paper.

2. Related work

Large-scale data warehousing technologies abound in the literature, many of which are based on Google's MapReduce [4] and Bigtable [5], such as Hadoop [6] and HBase[7], as well as NSA's Accumulo[8], which added cell-level security, increased fault tolerance (FATE), and a novel server-side processing paradigm [9]. Tools such as Google's Drill [10], and the Apache Software Foundation implementation Dremel [11], promise SQL-like interactive querying over these Bigtable-backed frameworks. Hive [12] and Pig [13] both aim to permit definition of analytics over arbitrarily formatted data in Hadoop [6], while Cascading [14] takes a slightly more engineer-centric approach to definition of analytics over Hadoop.

Some of the more common projects in the streaming analytics space are IBM's InfoSphere Streams [1], and the open source Storm [2], developed by BackType and now an Apache Incubator project. Others include Yahoo!'s S4 [3] (also in the Apache Incubator), which offers an agent-based programming model. This makes deployment scenarios and performance prediction somewhat more challenging than Storm and Streams, which offer a lower-level abstraction. Esper [15] provides a cross-platform API for Java and .NET, and Microsoft's StreamInsight [16] product offers tight integration with Microsoft SQL Server.

Most of these technologies facilitate execution of an analytic over a single paradigm, be it online or offline. Recently, researchers have begun to translate offline analytics into an online paradigm. SAMOA [17] aims to enable Machine Learning using a streaming processing paradigm to both validate and update models in near-real-time. AT&T Research, as part of their Darkstar project [18], have constructed a hybrid stream data warehouse, DataDepot [19]. This uses online techniques to perform analysis on data as it arrives at the data warehouse, updating the contents of the bulk data store in the process. The closest research to CRUCIBLE to date has been in IBM DEDUCE [20], which defines code for MapReduce using SPADE (Stream Processing Application Declarative Engine), the programming language used in early versions of InfoSphere Streams. This permits a unified programming model (*e.g.*, allowing use of common operators), but does not offer any direct execution equivalence between a MapReduce job and an equivalent Streams SPADE job. Furthermore, SPADE is now deprecated in favour of SPL (Stream Processing Language).

CRUCIBLE builds on the most desirable attributes of these approaches in order to offer a single framework for developing secure analytics to be deployed at scale on state of the art multi-tenancy on- and off-line data processing platforms. It offers a similar programming model and approach to task parallelism as the likes of Storm and Streams, while offering consistent execution semantics across both on- and off-line data. It includes a semi-automated framework for management of security labels, and permits the application of these labels equivalently across data sources.

3. CRUCIBLE system

In order to facilitate the creation of advanced analytics for on- and off-line distributed execution, the CRUCIBLE DSL makes use of a higher level language abstraction than typical analytic frameworks, such as [2], [11], or [14]. This enables a degree of portability that is not typically achievable under other schemes; an engineer may write their analytic once, in a concise high-level language, and execute across a variety of paradigms without knowledge of runtime-specific implementation details. In addition, the user is afforded the ability to exploit an array of best-in-class runtime models for the execution of CRUCIBLE code.

Furthermore, this approach seeks to free domain specialists from concerns of correctness and security. The CRUCIBLE runtimes are responsible for ensuring that analytics are run with equivalent execution semantics, through adherence to CRUCIBLE's execution model, thus providing assurances of cross-platform correctness. The domain-specific nature of the CRUCIBLE language permits the user a greater degree of confidence that the analytic they *intend* is the analytic they have

written. As well as providing assurances regarding functional correctness, the automated application of security labelling frees the user from having to ensure they have not violated the security caveats associated with the data they are using.

A risk organisations face when creating a suite of analytics is the constantly evolving state-of-the-art in analytic frameworks. CRUCIBLE can help to mitigate this risk, as the "porting" of an entire suite of analytics becomes a matter of introducing a new CRUCIBLE runtime for the new framework – provided the runtime adheres to CRUCIBLE's execution model, portability of correctness can be assured with confidence.

3.1. CRUCIBLE DSL

CRUCIBLE's DSL is built on the XText [21] language framework. It makes use of XBase, which is an embeddable version of the XTend JVM (Java Virtual Machine) language. CRUCIBLE's DSL provides a syntactic framework for modelling Processing Elements, using XBase for the processing element (PE) logic.

CRUCIBLE transpiles a topology (a collection of connected PEs) into idiomatic Java based on the CRUCIBLE PE Model (see Fig. 1). This is in contrast to many other JVM languages, such as [22], which directly compile into unreadable bytecode. Compiler support is used to provide syntactic sugar for accessing global shared state and the security labelling mechanism, which are discussed in more detail in Section 3.4.

At a high level, a CRUCIBLE analytic (such as in Listing 1) is structured similarly to a Java code file; it consists of a package declaration, a set of imports, and then one or more classes. In the CRUCIBLE DSL, each `process` models a class, with a name and an optional superclass. These classes are referred to as Processing Elements, or PEs for short. The body of a process is divided into a set of unordered blocks:

- `conf` – Compile-time configuration constants. The calculation of these may involve an arbitrary expression.
- `state` – Runtime mutable state; shared globally between instances of this PE. These variables may be declared `local`, in which case no global state is utilised for their storage (see Section 3.3).
- `outputs` – Declaration of the named output ports from the `process`.
- `input` – A block of key/value pairs mapping the qualified name of an output (in the form `ProcessName.OutputName`) to a block of code to execute upon arrival of a tuple from that port.

Listing 1. An Example CRUCIBLE Topology fragment, counting the frequency of characters in the input.

```
1  package eg.counter
2
3  import crucible.lib.pe.FileSource
4  import crucible.lib.pe.FileSink
5
6  process Source extends FileSource {
7     config : {
8       Filename = '/usr/share/dict/words'
9       ReadLines = false // Read chars, not lines
10    }
11    outputs : [FileLine, FileCharacter]
12 }
13
14 process Filter {
15    config : int N = 1500000 // For TopN calculation
16    state : int seen = 0
17    outputs : [Keys]
18    input : Source.FileCharacter -> {
19      if ((seen) >= N) {
20        Keys.emit('done' -> true, 'key' -> Character::MIN_VALUE)
21      } else if (Character::isLetter(character)) {
22        seen = seen + 1
23        Keys.emit('key' -> Character::toUpperCase(character),
24                  'done' -> false, 'total' -> seen)
25      }
26    }
27 }
28
29 process CountingWriter extends FileSink {
30    output : Results
31    state : counts = ('A'.charAt(0) .. 'Z'.charAt(0))
32           .toInvertedMap[ new AtomicInteger ]
33    config : Filename = 'counts.txt'
34    input : Filter.Keys -> {
35      if (done) {
36        log.info(counts.toString)
37        Results.emit('total' -> total, 'counts' -> counts as Map,
38                     'tstamp' -> System::currentTimeMillis)
39      }
40      counts.get(key.charValue as int)?.incrementAndGet
41    }
42    input : CountingWriter.Results -> super
43 }
```

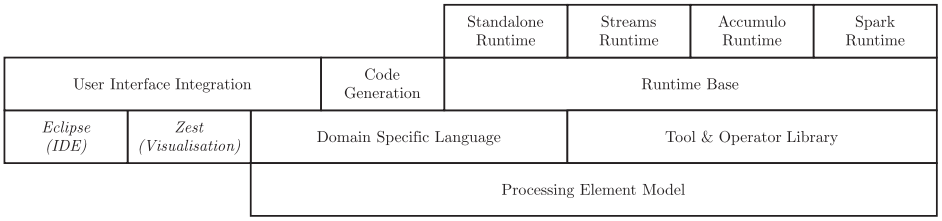

Fig. 1. Components of the CRUCIBLE system. Entries in *italics* are external dependencies.

3.2. Message passing

CRUCIBLE PEs communicate using message passing; a call to *OutputName*`.emit(...)` causes all subscribers to that output to receive the same message. No guarantees are given about the ordering of messages interleaved from different sources. Messages are emitted as a set of key-value pairs, encoded as a single tuple. At compile time CRUCIBLE performs type inference on all of the `emit` calls in the topology to generate a correctly typed and named `receive` method interface on each subscriber; the key of an item in the tuple is used as the parameter name on the method.

3.3. Global synchronisation & state

CRUCIBLE's global synchronisation and shared state components make use of the `GlobalStateProvider` and `LockingProvider` implementations which are injected at runtime. As discussed, `state` variables exist in global scope if they are not marked `local`. Thus, if multiple instances of a PE are run simultaneously, they will share any updates to their state; these changes are made automatically. This mechanism is applied without any guarantees about transactional integrity, which in limited circumstances is acceptable, *e.g.*, when sampling for 'a recent value'.

In those circumstances which require distributed locking, an `atomic` extension method is provided to take a lock on a given state element, and apply the given closure to the locked state. The behaviour of this is similar to Java's `synchronized` keyword, with two key distinctions. The first of these is that the locking is guaranteed across multiple instances of a PE within a job, even across multiple hosts. The second distinction is that the `atomic` method may be applied to multiple objects by locking a list of variables, *e.g.*, `#[x, y, z].atomic[...]`, in which case all locks are acquired before invoking the closure. A fixed ordering of locking and unlocking is applied, as well as a protocol lock, to ensure that interleaved requests across critical regions do not deadlock.

3.4. Security labelling

CRUCIBLE's Security Labelling system is motivated by the need to cope with complex access control requirements in multi-tenancy environments. For example, the provenance or classification of data may need to be tracked on a cell level in order to determine the visibility of a datum for a user. Ensuring that these visibilities are tracked consistently is a challenge that requires a great deal of attention to detail throughout the evolution of an analytic system.

CRUCIBLE's labelling protocol is built on the concept of cell-level visibility expressions, similar to those described by Bell and La Padula [23]. An expression is given as a conjunction of disjunctions across named labels. For example, the expression "$A\&(B\|C)$" requires that a user is authorised to read the *A* label, as well as either *B* or *C*. If they lack sufficient authorisation, they should not even be aware of the existence of that cell.

In practise, this concept is implemented by declaring an empty security label for every variable in the system. This label is accessible to the user by calling the `label` extension method on a variable. A user may add to a label using the `+=` operator. For example, the label associated with the `x` variable is expanded by either calling `x.label += "A | B"` (literal expansion), or `x.label += y.label` (expansion by label reference). More formally, consider a label function λ, and a label expansion function ϵ:

$$\begin{aligned} &\lambda(a) : \text{Label for identifier } a \\ &\epsilon(a,b) : \lambda_1(a) = \{\lambda_0(a), \lambda(b)\} \end{aligned} \tag{1}$$

Labelling of object-oriented method invocation makes the assumption that the receiver's state may be mutated by the supplied arguments. Therefore:

$$c.foo(d,e,f) \Rightarrow \begin{cases} \epsilon(c,d) \\ \epsilon(c,e) \\ \epsilon(c,f) \end{cases}$$
$$\lambda_1(c) = \{\lambda_0(c), \lambda(d), \lambda(e), \lambda(f)\} \tag{2}$$

Assignment of a value to a non-final Java variable (*e.g.*, `g = h`, where `h` is any expression; not to be confused with `g.label = h.label`) requires clearing the contents of its label prior to expansion, as accumulated state is discarded:

$$g = h \Rightarrow \lambda_1(g) = \emptyset \tag{3}$$

If the right expression (`h`) contains any identifiers, expansion must occur;

$$\forall(i) \in h, \mathit{identifier}(i) \Rightarrow \epsilon(g,i)$$
$$\lambda_2(g) = \{\lambda(i_0) \ldots \lambda(i_n)\} \tag{4}$$

As objects may contain mutable state, when a label for `x` expands to encompass `y`, and `y` is later expanded, `x`'s label must include the additions to `y`:

```
1 # Ass: y.label == ''
2 x.label += 'foo'
3 x.doSomething(y)
4 y.label += 'bar'
5 x.label == 'bar&foo'
```

$$\begin{aligned} \lambda_0(y) &= \emptyset \\ \lambda_0(x) &= \{\text{“}foo\text{”}\} \\ \epsilon(x,y) & \\ \lambda_1(y) &= \{\text{“}bar\text{”}\} \\ \lambda_2(x) &= \{\text{“}bar\text{”}, \text{“}foo\text{”}\} \end{aligned} \tag{5}$$

3.4.1. Application of labelling

This labelling requires support from the CRUCIBLE compiler to transform invocations of the tuple emission method, `emit(Pair<String,?> ...tuple)`, into invocations of the form `emit(Pair<SecurityLabel, Pair<String,?>> ...tuple)`. Note that in the Java type system this has the same type erasure as the original method. Concordantly, when generating the signature for a `receive` method, the compiler interleaves parameters with their labels. Thus, an interface of `(String, Integer)` instead becomes `(Security Label, String, SecurityLabel, Integer)`. Listing 2 shows a simple CRUCIBLE fragment, designed to illustrate the automated application of security labelling in practice, while Listing 3 shows what this code would transpile to after processing by the CRUCIBLE compiler. Note in this listing how the state declaration of

Listing 2. CRUCIBLE fragment for calculating the mean of results from Listing 1.

```
process Mean {
 state : int sum = 0
 output : RunningAverage
 input : Filter.Keys -> {
    if (total % 100 == 0) {
       RunningAverage.emit('mean' -> sum / total)
       sum = 0
    }
    sum = sum + key.charValue as int
 }
}
```

Listing 3. Fragment of transpiled code from Listing 2.

```
protected int $_sum = 0;
protected final SecurityLabel sum$label = new SecurityLabel();

public void receive$Filter$Keys(
        final SecurityLabel done$label, final boolean done,
        final SecurityLabel key$label,  final char key,
        final SecurityLabel total$label, final int total) {
  if (((total % 100) == 0)) {
    int _divide = (this.getSum() / total);
    Pair<Object,Object> _mappedTo = Pair.<Object, Object>of(
        "mean", Integer.valueOf(_divide));
    this.RunningAverage.emit(Pair.<SecurityLabel, Object>of(
        new SecurityLabel(sum$label, total$label), _mappedTo));
    this.setSum(0);
    sum$label.$clear();
  }
  char _charValue = Character.valueOf(key).charValue();
  this.setSum(this.getSum() + ((int) _charValue));
  sum$label.expand(key$label);
}
```

`sum` is given an instance variable, `$_sum`, and a `SecurityLabel`, `sum$label`. Furthermore, the labelling rules described above are applied consistently in the code: as `sum` is modified using the value of `key`, the `sum$label` is expanded to encompass the instance of `key$label` passed from the upstream PE (line 22, per Rule 2). Similarly, when the value of `sum` is cleared (Listing 2 Line 7), `sum$label` is also cleared (Listing 3 Line 16), per Rule 3. As the assignment is to an absolute value with no associated label, no further expansion is required at this point (as in Rule 4).

Thus, if a sequence of tuples were emitted to the `Mean` PE from `Filter.Keys` such as the following, the given output would occur from `Mean.RunningAverage` (the below table uses [..] notation to denote the security label associated with a value). In this example, keys are labelled as to whether they are a consonant ("C"), or a vowel ("V"). The total seen is always labelled "S", for Sum.

`Filter.Keys`			→	`Mean.RunningAverage`
`done`	`key`	`total`		`mean`
false []	G [C]	98 [S]		0.72 [C & S]
false []	H [C]	99 [S]		1.44 [C & S]
false []	I [V]	100 [S]		0.73 [V & S]
false []	J [C]	101 [S]		1.46 [V & C & S]

It is important to note that due to CRUCIBLE's close integration with the JVM, this mechanism should not be considered secure for arbitrary untrusted code; it aims only to assist the security-conscious engineer by making it easier to comply with security protocols than to ignore them.

3.5. CRUCIBLE runtimes

CRUCIBLE offers three key runtime environments for execution of analytics transpiled from the DSL source. Fig. 2 shows how classes in the model interact; instances of many of these classes (shaded) are injected at runtime using [24], permitting the behaviour of the topology to be integrated with the relevant runtime engine without changes or specialisation in the user code.

3.5.1. Standalone processing

The first, and simplest, runtime environment is designed for easy local testing of a CRUCIBLE topology, without any need for a distributed infrastructure. This Standalone environment simply executes a given topology locally, in a single JVM, relying heavily on Java's multithreading capabilities. Locking and global state are provided based on this in-JVM assumption.

Message passing is performed entirely in-memory, using a singleton `Dispatcher` implementation with a blocking concurrent queue providing backpressure [25] in the event that some PEs are slower than others. This prevents the topology from using an excessive amount of memory.

3.5.2. On-line processing

IBM's InfoSphere Streams product forms the basis of CRUCIBLE's streaming (on-line) runtime engine. An extension to the CRUCIBLE DSL compiler generates a complete SPL (IBM's Streams Processing Language) project from the given topology. This

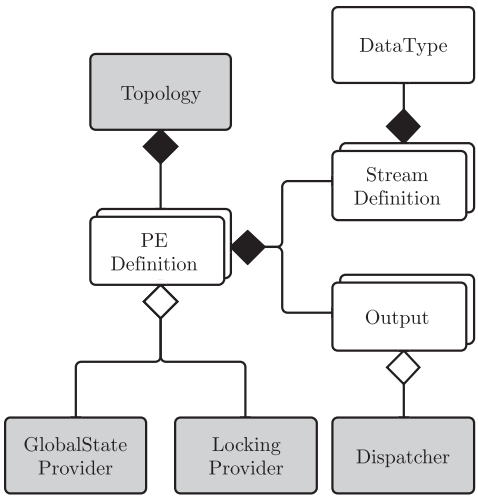

Fig. 2. CRUCIBLE model composition diagram, showing the composition of the core model (white) and the runtime injectable components (grey).

project can be imported into InfoSphere Streams Studio; it consists of the required project infrastructure (including classpath dependencies), and a single SPL Main Composite describing the topology. Each SPL PE in Streams is an instance of the `CruciblePE` class. This class handles invocation of the `receive$...` tuple methods, dispatch between Streams and the `CruciblePE`s, and tuple serialisation.

There is a one-to-one mapping between tuples emitted in CRUCIBLE and tuples emitted in Streams. Each key in a CRUCIBLE tuple has a defined field in a Streams tuple, and all keys are transmitted with each emission. Keys are interleaved with their security labels, such that the label for a key always precedes it. Tuple values are converted between Streams and CRUCIBLE using an injected serialisation provider; the default implementation of this leverages Kryo for time and space efficiency [26,27], but it would be trivial to add, for example, a Protocol Buffers-based implementation if interoperability with external systems were required. Security Labels are *not* serialised through Kryo, in order to facilitate their inspection by debug tooling on the Streams instance. Security Labels are written as `rstring` values, while all others are serialised as a `list<int8>` (representing an immutable array of bytes).

Each of these `CruciblePE` instances could potentially be scheduled into separate JVMs running on different hosts, according to the behaviour of the Streams deployment manager. Manual editing of the SPL, *e.g.*, to use SPLMM (SPL Mixed Mode, using Perl as a preprocessor), can be used to split a single PE across multiple hosts. The injectable global synchronisation primitives discussed in Section 3.3 must be enabled on the runtime to facilitate this form of data-parallelism.

3.5.3. Off-line processing

The mapping from CRUCIBLE's execution model to Accumulo for off-line processing is more involved. In order to exploit the data locality and inherent parallelism available in HDFS, while maintaining the level of continuous insight offered by Streams, the Accumulo runtime makes use of Accumulo Iterators [9]. An `Iterator` may scan multiple tablets in parallel, and will stream ordered results to the `Scanner` which invoked the iterator. CRUCIBLE makes use of this paradigm by spawning a `CrucibleIterator` for each PE in the topology, along with a multithreaded `Scanner` to consume results. Each `CrucibleIterator` may be instantiated, destroyed, and re-created repeatedly as the scan progresses through the data store.

Each `CrucibleIterator` is assigned to its own table, named after the `UUID` of the Job and the PE to which it refers. Values map onto an Accumulo Key by using a timestamp for the Row ID, the Source PE of a tuple as Column Qualifier, and the emitted item's key as Column Key. Column Visibility and Security Label are mapped directly onto their Accumulo equivalents, making efficient use of native constructs.

In this way, the `CrucibleIterator` can invoke the correct `receive` method on a PE, by collating all (*key, value, label*) triples of a given `RowID`. By mapping CRUCIBLE Security Labels onto Accumulo Visibilities, all message passing data (and final results) are persisted to HDFS with their correct labels: external Accumulo clients may read that state, provided they posess the correct set of `Authorization`(s).

CRUCIBLE's `AccumuloDispatcher` takes tuples emitted by a PE, and writes them to the tables of each subscriber to that stream, for the relevant `CrucibleIterator`s to process in parallel. The final component is the multithreaded `Scanner`, which restarts from the last key scanned, thus ensuring that the Accumulo-backed job fully processes all tuples in all tables. This flow is presented in Fig. 3, for clarity.

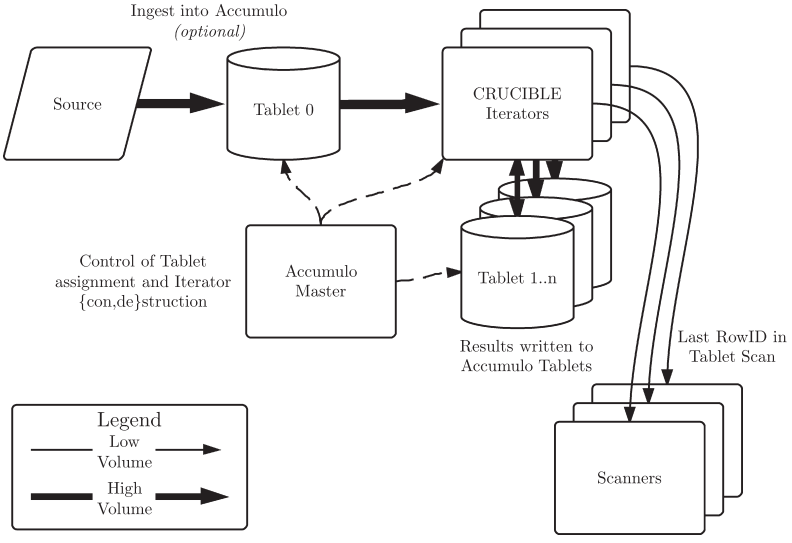

Fig. 3. CRUCIBLE Accumulo runtime message dispatch, demonstrating how Scanners are used to pull data through a collection of custom Iterators to analyse data sharded across Accumulo Tablets.

3.6. Standard library

The last CRUCIBLE component is the standard library. This includes the components necessary for operation of the above runtimes, along with a set of base PE implementations to simplify creation of CRUCIBLE topologies. These provide examples of data ingest from a variety of sources, such as from the APIs of Flickr and Twitter, along with primitives to read and write file data. An XPath PE is valuable for extracting data from XML. Work is underway to expand this library to include operators for parallel JOINs, Bloom-Filters, and serialisation to/from common data formats such as JSON.

This library is implemented in standard Java, and no special infrastructure is required to extend it. It is intended that users of CRUCIBLE may extend this library with custom PEs, or publish their own, simply by writing Java which conforms to a given interface, and making it available on the classpath. For single-use Java operators this may even be done within the analytic's Eclipse IDE project – the compiler will load and integrate the operator automatically.

4. CRUCIBLE runtime optimisation

Section 3.5 described the implementation of the three core CRUCIBLE runtimes. The work in [28] demonstrated near linear scaling of their performance over growing input sizes. However, this work also demonstrated that they lacked sufficiently strong performance when compared to hand-written implementations. A series of significant enhancements and optimisations have been implemented in each of these runtimes, in order to improve time-to-solution performance. The experimental results described below were collected on the same specification of system described for the original CRUCIBLE experimental results, using the same benchmark. These results are, therefore, directly comparable. This simple CRUCIBLE benchmark topology was written to count the frequency of letter occurrence in a dictionary (akin to Listing 1), limited to the top N results, where N is the problem size. There is a key distinction between CRUCIBLE and the native implementations here; as the native environments lack support for security labelling, only the CRUCIBLE runtimes track the per-cell security labels. The results were collected on a small development cluster, consisting of three Accumulo Tablet Servers, one Master, and three Streams nodes. Each node hosts two dual-core 3.0 GHz Intel Xeon 5160 CPUs, 8 GB RAM, and 2x1GbE interfaces.

4.1. Standalone processing

The standalone processing environment provides an ideal test bed for general optimisations to the CRUCIBLE framework, due to its lack of complex IPC. Fig. 4 shows the proportion of the runtime spent in various functional sections of the code for each of the standalone runtime models implemented. Figs. 4 and 9 were generated using an instrumented JVM for the maximum problem size. They omit the warm-up time, in order to illustrate the proportional function runtimes for an analytic during the bulk of its execution. It is important to note that these data do not capture the proportion of time spent blocked or context switching, and are thus not sufficient on their own for comparing the absolute performance of the runtimes.

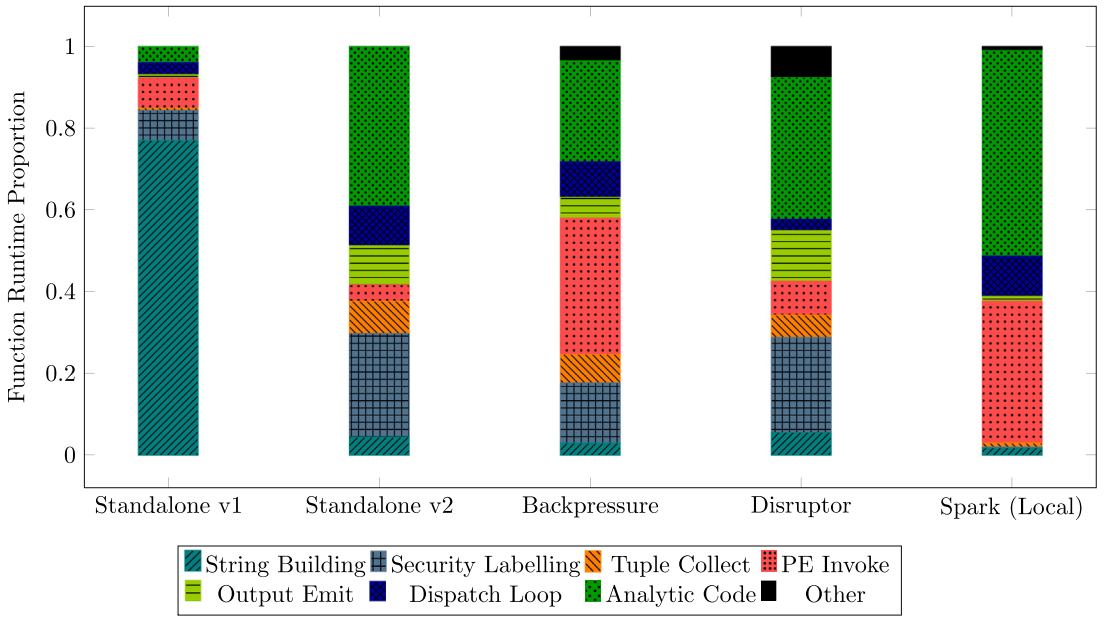

Fig. 4. Function runtime breakdown across Standalone Dispatchers.

The original version of the standalone runtime (`Standalone v1`) spends over 75% of its wall time building strings, either naïvely for logging purposes or for analytic output. Additionally, approximately 7.5% of the runtime is spent examining PE configuration and building data structures to invoke the topology's PEs. A combination of smarter log management, and better code generation, permits the `Standalone v2` entry to spend a significantly higher proportion of its time ($\approx 40\%$ rather than $\approx 4\%$) performing the actual analysis. This new code generation offers a set of guarantees to the CRUCIBLE runtime that permit further compile-time reasoning about the values that are emitted and received. For example, it is now guaranteed that parameters on the `emit` and `receive` interfaces of a given PE are equivalently ordered when the code is generated. This allows the runtime to perform far less manipulation and reasoning about its PE configuration in memory when invoking a PE.

With a more optimised Standalone environment, the importance of thread utilisation becomes increasingly apparent. Fig. 5 shows how the set of workers in the Standalone v2 dispatcher (top chart) spend a significant amount of time switching in and out of a runnable state; none of the Worker threads (responsible for receiving submitted tuples and invoking the relevant PE) show full CPU utilisation. Two further experiments were conducted based on these results. The first of these,

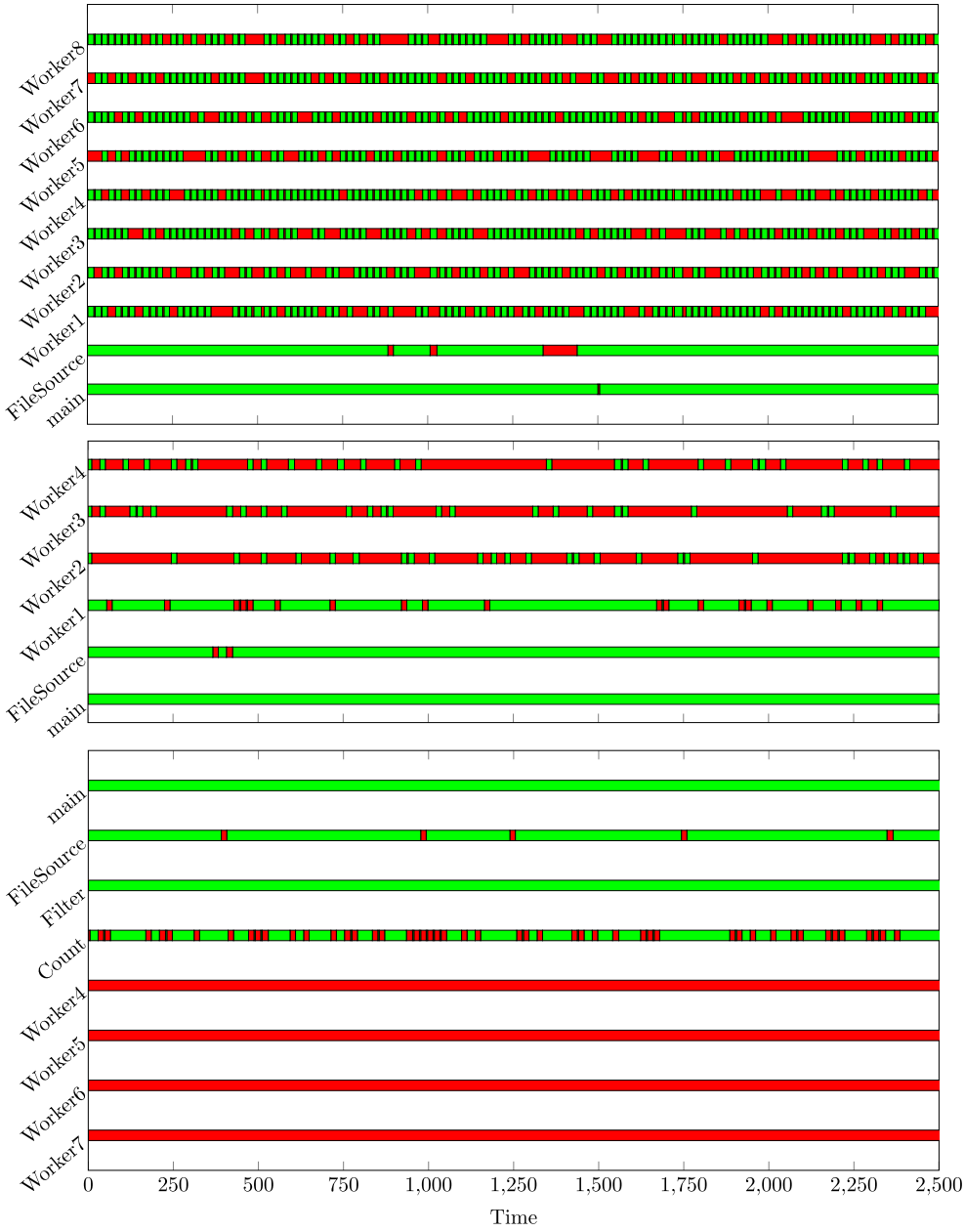

Fig. 5. Thread utilisation in the Standalone, Backpressure, and Disruptor Dispatchers respectively.

illustrated in the middle chart of Fig. 5, introduced the use of backpressure [25] to slow down PEs that were producing faster than downstream PEs could consume (thus reducing the probability of resource starvation). This shows much better thread utilisation, but does not make adequate use of the multi-core architecture on which it runs.

The final, and best performing, standalone Dispatcher implementation made use of the LMAX Disruptor [29], detailed in the bottom chart of Fig. 5. It is noteworthy that the Disruptor Dispatcher scheduled each PE to its own thread consistently, and significantly reduced the amount of context switching by permitting the threads to run truly concurrently whenever there was data available. The superior thread utilisation of the Disruptor Dispatcher is borne out in the runtime results of Fig. 6, demonstrating a speedup of over 16× of a Disruptor-based runtime model over the original Standalone model. Whereas the original model suffered a performance penalty of 526× over the native implementation, the Disruptor model is only 32× slower.

4.2. On-line processing

In order to better understand the costs involved in the existing CRUCIBLE SPL execution model, an SPL topology was instrumented to measure the latency introduced by tuple I/O. The Native SPL results show the latency in passing a message

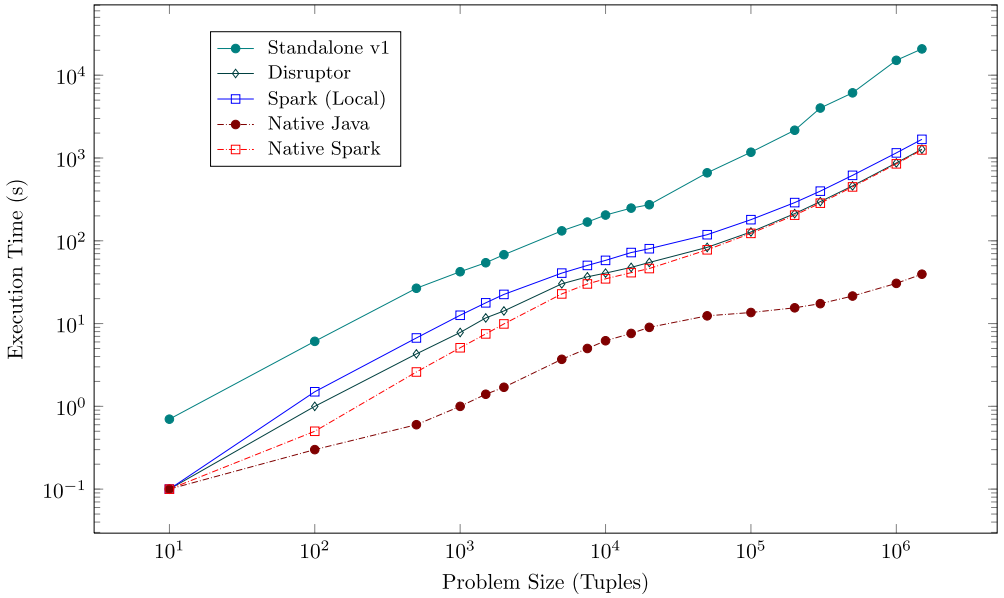

Fig. 6. Scalability comparison of CRUCIBLE Standalone runtimes and native implementations.

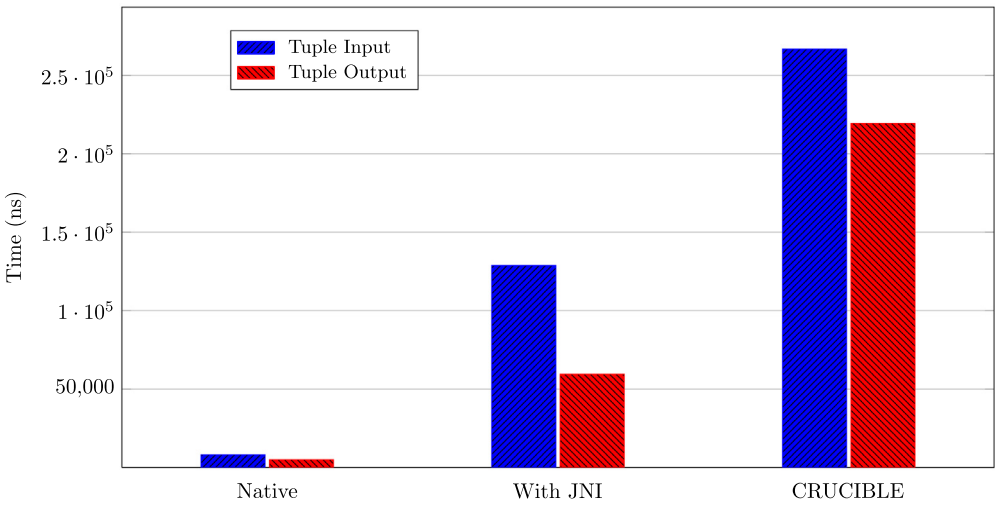

Fig. 7. SPL Tuple I/O Instrumentation.

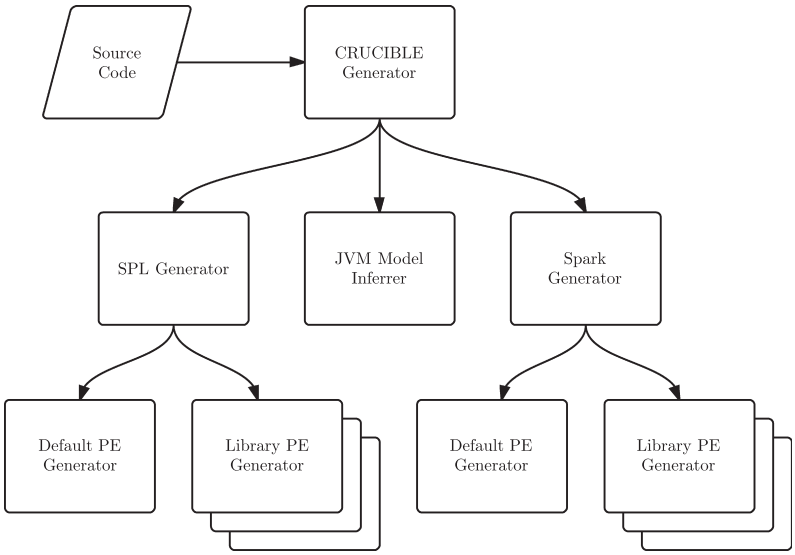

Fig. 8. CRUCIBLE Code Generation Hierarchy.

into or out of a PE written entirely in SPL. JNI (the Java Native Interface) does not involve any CRUCIBLE code; it simply measures the latency introduced by causing tuples to be passed from the Streams SPL interface into the Streams Java interface. This is a necessary precondition for execution of CRUCIBLE's Java target code. The final results include the SPL and JNI latencies, as well as those introduced by transcoding and converting data types between Streams Java tuples and tuples in CRUCIBLE, as well as execution of the PE invocation logic. The results of these measurements are summarised in Fig. 7; the JNI bridging process is responsible for a considerable proportion of the latency in invoking a CRUCIBLE PE.

In order to minimise the impact of Streams' JNI latency, it is necessary to improve CRUCIBLE's generation of the SPL target to make best possible use of native SPL operators. For example, many of the CRUCIBLE library functions exist natively within SPL (*e.g.*, file sources and sinks, or timed "beacon" emitters) as higher performance variants of the Java code. By introducing a new pluggable code generation architecture, CRUCIBLE enables library code developers to hook into the generation engine and create *runtime specific* variants of a given PE, provided they ensure that the generated set of runtime-specific PEs are properly typed and named.

This pluggable code generation system offers each generator for a pair of PE class/runtime environment the *opportunity* to generate code for a given instance of a PE by simply being loaded on the classpath for the IDE. The generators are sought using their annotation: `@Generate(target = SomePE.class, type = "SPL")` indicates that the annotated

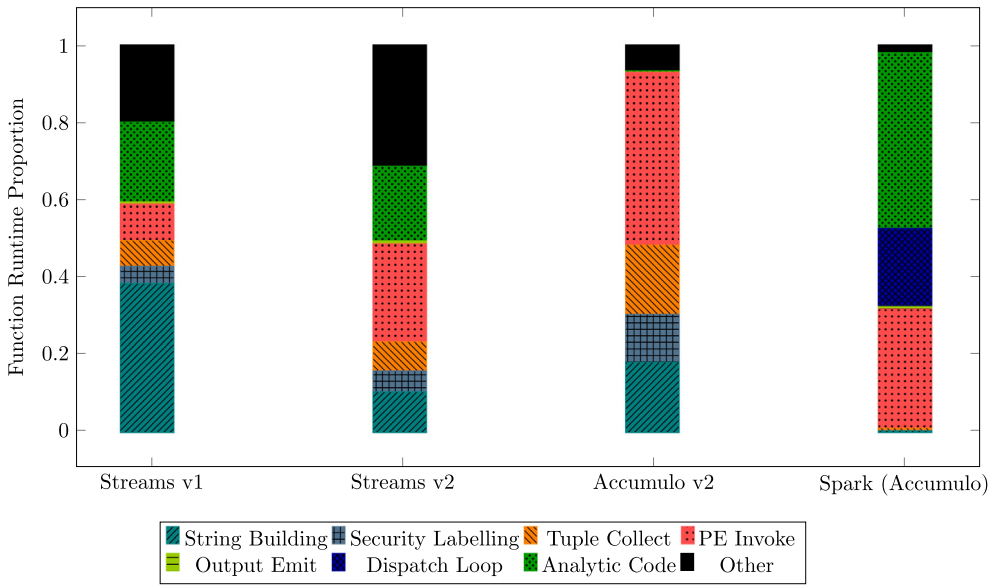

Fig. 9. Function runtime breakdown across On- and Off-line Dispatchers.

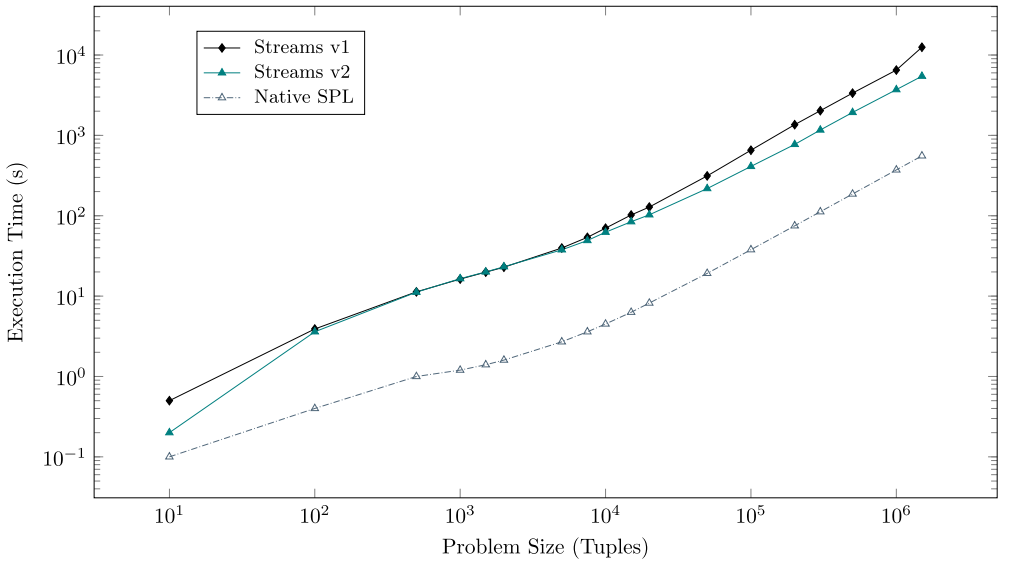

Fig. 10. Scalability Comparison of CRUCIBLE Online Runtimes and Native Implementations.

class is a generator for `SomePE` in the `SPL` runtime. The integration of the various code generation components in the new architecture is highlighted in Fig. 8. The existing CRUCIBLE Java code generation is performed by the same JVM Model Inferrer as previously; the new architecture simply adds the pluggable generators on the bottom layer. For example, Listing 4 shows the original output of the CRUCIBLE SPL generator; note the application of a `CruciblePE` instance to each node in the SPL graph. Listing 5 shows the same analytic compiled under the new code generation mechanism. The type signatures are identical, but native code is instead generated to support the file source and sink PEs from the CRUCIBLE topology without altering the semantics of the tuple processing. Fig. 9 details how the use of some simple SPL primitives (notably file source and sink operators) has impacted the breakdown of function runtime in Streams.

Fig. 10 illustrates the 2.3× speedup that more advanced SPL generation has allowed on the existing CRUCIBLE benchmark – the performance of CRUCIBLE transpiled for InfoSphere Streams, once 22× slower than a native implementation, is now under 10× slower. The nature of these improvements is such that they can offer even greater speedups as the topology becomes more complex, utilising more SPL library functions.

Listing 4. CRUCIBLE Streams v1 SPL Generation.

```
composite Process
{
    type
        FilterCount__Results__Type = tuple<rstring counts__label,
            list<int8> counts, rstring total__label, list<int8> total,
            rstring tstamp__label, list<int8> tstamp>;
        Source__FileLine__Type = tuple<rstring done__label,
            list<int8> done, rstring line__label, list<int8> line>;
        Source__FileCharacter__Type = tuple<rstring character__label,
            list<int8> character, rstring done__label, list<int8> done>;

    graph
        (stream<Source__FileLine__Type> Source__FileLine;
         stream<Source__FileCharacter__Type> Source__FileCharacter
        ) = CruciblePE()
        {
            param
                peClass : 'freq.Source';
                configModule : 'freq.ProcessConfigurationModule';
        }

        (stream<FilterCount__Results__Type> FilterCount__Results) =
            CruciblePE(Source__FileCharacter)
        {
            param
                peClass : 'freq.FilterCount';
                configModule : 'freq.ProcessConfigurationModule';
        }

        () as Write = CruciblePE(FilterCount__Results)
        {
            param
                peClass : 'freq.Write';
                configModule : 'freq.ProcessConfigurationModule';
        }

}
```

Listing 5. CRUCIBLE Streams v2 SPL Generation.

```
composite Process
{
   type
      FilterCount__Results__Type = tuple<rstring counts__label,
         list<int8> counts, rstring total__label, list<int8> total,
         rstring tstamp__label, list<int8> tstamp>;
      Source__FileLine__Type = tuple<rstring done__label,
         list<int8> done, rstring line__label, list<int8> line>;
      Source__FileCharacter__Type = tuple<rstring character__label,
         list<int8> character, rstring done__label, list<int8> done>;

   graph
      stream<blob value> Source__File__Source = FileSource()
      {
         param
            file : '/usr/share/dict/words';
            format : block;
            blockSize : 1u;
      }
      stream<Source__FileCharacter__Type> Source__FileCharacter =
         Functor(Source__File__Source)
      {
         output Source__FileCharacter :
            character__label = '',
            character = (ustring)convertFromBlob(value),
            done__label = '',
            done = false;
      }


      (stream<FilterCount__Results__Type> FilterCount__Results) =
         CruciblePE(Source__FileCharacter)
      {
         param
            peClass : 'freq.FilterCount';
            configModule : 'freq.ProcessConfigurationModule';
      }

      () as Write = FileSink(Count__Results)
      {
         param
           file     : '/nfs/tmp/count.txt';
           append : true;
           format : txt;
      }
}
```

4.3. Off-line processing

Fig. 11 reveals problems with interfacing Accumulo with the original runtime model: even after the optimisations described in Section 3.5.1, the large number of Tablet Read-Ahead threads show drastic under-utilisation for the workload (see Fig. 3 for an overview of how the PollingScanner instances interact with Accumulo's tablet servers; these Tablet Read threads host the custom CRUCIBLE Iterators described in the figure). Furthermore, the process of swapping an Accumulo Iterator out of a read-ahead thread is such that it forces a rebuild of the iterator's state when it is swapped back in. The impact of this can be clearly seen the `Accumulo v2` column of Fig. 9; a vanishingly small proportion of time is spent processing the actual analytic, with the vast majority being spent in invocation methods – including Iterator construction and state retrieval.

These results reveal that the Accumulo Iterator model is incompatible with performing heavy computation and message passing using the Accumulo table interfaces at scale. Instead, these optimisations introduce (i) the use of Apache Spark [30] for execution of CRUCIBLE analytics over data in either Accumulo or native HDFS. In support of this, CRUCIBLE introduces (ii) a new library PE with close integration with the Spark Code Generator: the `DataSource`. A CRUCIBLE `DataSource` is an abstraction of the concept of a source, identified by a URN, with a fixed set of outputs per tuple. The precise code used to retrieve tuples from the source are determined by the runtime that is loaded; it may be an Accumulo table, a file in HDFS, or a streaming source, *e.g.*, a network socket.

The CRUCIBLE Spark runtime pulls data from the relevant `DataSource` through a series of Spark `map` operations to apply the directed graph of CRUCIBLE PEs to the full dataset. The precise scheduling of these operations is determined optimally by the Spark runtime engine. Each `map` operation emits an RDD (Resilient Distributed Dataset; an abstraction for a collection of data managed by Spark) of pairs <`Output Name`, `Tuple Data`>, which is split along the key to create the relevant source of tuples for the next stage(s) of the analytic. This unique `DataSource`-based approach has the added advantage of providing an alternative execution paradigm for Standalone mode (the functional runtime breakdown of which is detailed in Fig. 4), as Spark may be run over in-memory datasets without the backing of a Hadoop RDD.

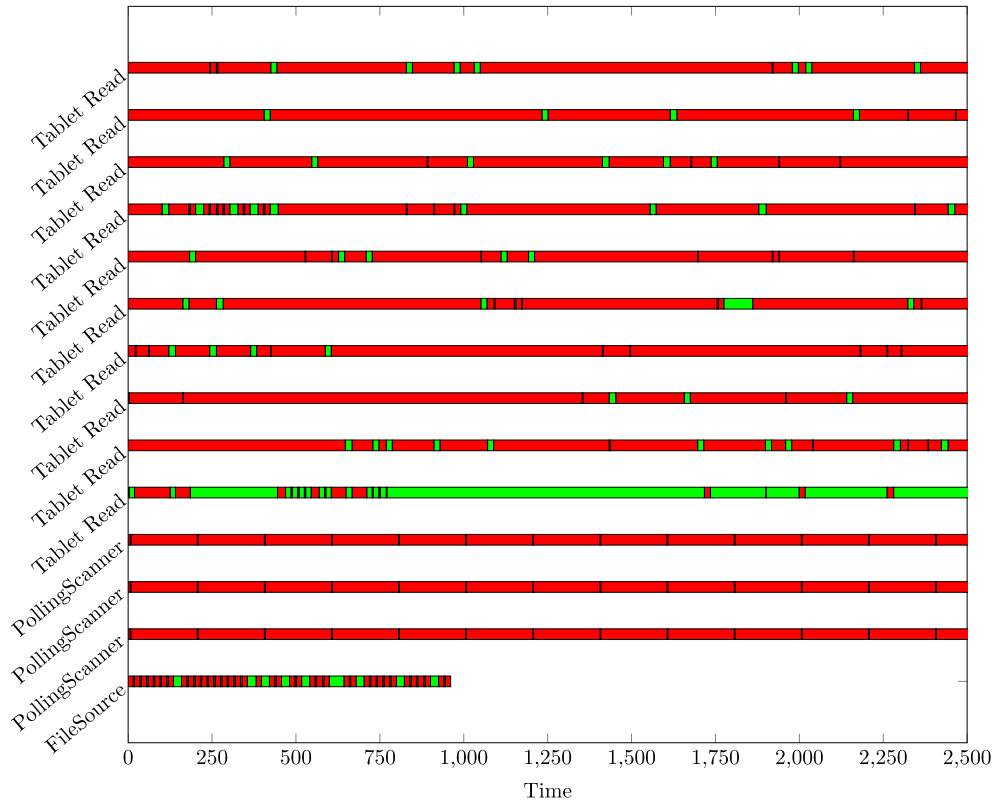

Fig. 11. Thread utilisation in the Accumulo (v2) Dispatcher.

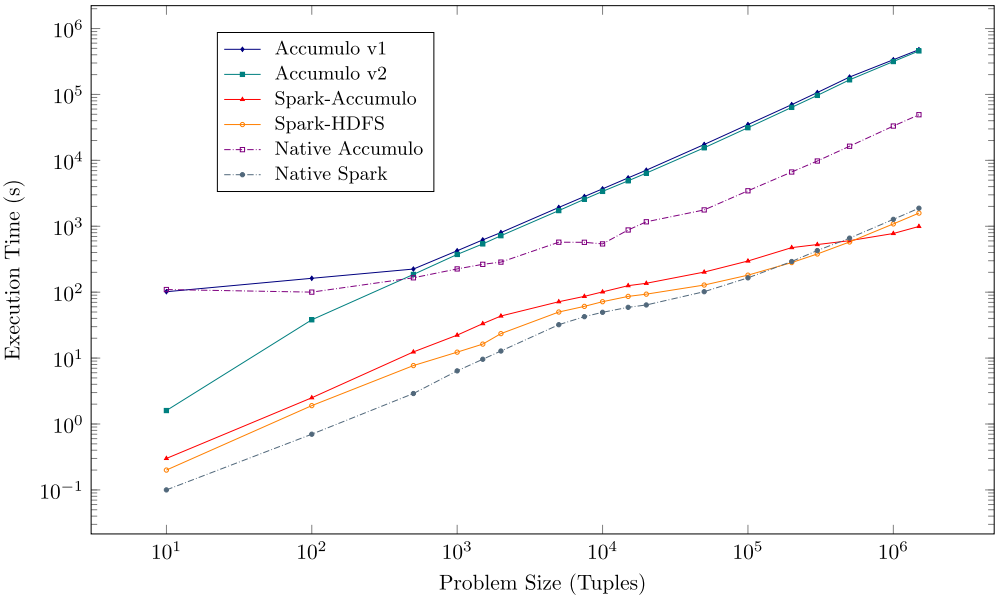

Fig. 12. Scalability comparison of CRUCIBLE offline runtimes and native implementations.

Fig. 9 demonstrates how superior the function breakdown is for the Accumulo-Spark runtime model compared to the original Accumulo Iterator interface, and Fig. 12 shows the significant performance enhancement in terms of time-to-solution that this offers; over 480× from the original Accumulo Iterator model to running Spark over Accumulo. Furthermore, the CRUCIBLE Spark runtime with the `DataSource` abstraction for the first time enables the processing of arbitrary Hadoop files

and text files in an equivalent yet scalable fashion using CRUCIBLE. At higher scales, CRUCIBLE's Spark-HDFS environment can even be seen to outperform a native implementation making use of the more expressive Spark builtins. Performing bulk analysis through the use of Accumulo Iterators with CRUCIBLE was approximately 10× slower than the equivalent native implementation; with Spark on HDFS files, this is now almost 1.2× faster than the native implementation used.

5. Future work

There are a number of avenues to be explored in future work. Ultimately, we intend to improve CRUCIBLE's usability through improved interfaces for planning and development of analytics. Alongside that, we intend to consider two key areas of future research; the CRUCIBLE DSL, and the runtimes.

In enhancing the CRUCIBLE DSL, we plan to introduce a number of new language features to further enhance CRUCIBLE's applicability and code reuse. For example, through the addition of compound PEs (a PE formed from a CRUCIBLE topology) and runtime PE reuse, it becomes possible to encode a commonly used set of PEs as a single PE in the topology, and reuse the results of its computation across multiple jobs. In this way, only the subsets of analyses which are different across given topologies must be computed separately, significantly enhancing the overall utilisation of cluster resources. In a similar vein, the capability to subscribe to results published from one job in another will permit a manually specified form of these efficiencies. Additionally, we plan to enhance in-IDE debug tooling, through the use of mock data sources, probes, and visualisations of the flow of data and propagation of security labels.

In addition to these language and IDE improvements, one promising avenue for future work is the investigation of novel approaches to the composition of analytic jobs in CRUCIBLE. Currently, this requires the manual authoring of program code. We intend to investigate further opportunities offered by CRUCIBLE's high-level abstraction to permit the creation of jobs by less technically competent domain experts.

In order to improve the runtime environments, we propose to investigate a number of alternative compilation strategies for topologies, in order to enable their execution of alternative architectures. This will enable workload-based optimisation for architecture selection - this could directly impact both deployment and procurement decisions. There are also plans to investigate PE Fusion and Fission Techniques to enhance data-level parallelism.

6. Conclusions

This paper has detailed the development of CRUCIBLE, a framework consisting of a DSL for describing analyses as a set of communicating sequential processes, a common runtime model for analytic execution in multiple streamed and batch environments, and an approach to automating the management of cell-level security labelling that is applied uniformly across runtimes. This work has served to (i) validate the approach that CRUCIBLE takes in transpiling a DSL for execution in a unified manner across a range on- and off-line runtime environments, as well as (ii) forming an investigation into techniques for deployment across these architectures. The work has demonstrated (iii) the application of analysis written in CRUCIBLE to data sources including HDFS files, Accumulo tables, and traditional flat files. The results presented demonstrate that the selection of runtime model for execution of CRUCIBLE topologies is critical; making a difference of up to 480x. The net result of these optimisations is (iv) a suite of best-in-class runtime models with equivalent execution semantics and a 14× performance penalty over the equivalent native hand-written implementations. The paper has demonstrated a number of valuable capabilities as a result of being based on a DSL; the tight integration of key language features, particularly security labelling and atomic operations, enables the implementation of sample analytics in one sixth the amount of code as the native implementations – a substantial improvement in engineering time, cost, and risk.

Further information on CRUCIBLE, including all available documentation, can be found at http://go.warwick.ac.uk/crucible.

Acknowledgments

This work was funded under an Industrial EPSRC CASE Studentship, entitled "Platforms for Deploying Scalable Parallel Analytic Jobs over High Frequency Data Streams". The lead author thanks their employer for funding the course of study that led to this paper.

References

[1] R. Rea, K. Mamidipaka, IBM InfoSphere Streams: Enabling Complex Analytics with Ultra-Low Latencies on Data in Motion, IBM, 2009.
[2] N. Marz, Storm. <http://storm-project.net/>, accessed 2013.
[3] L. Neumeyer, B. Robbins, A. Nair, A. Kesari, S4: distributed stream computing platform, in: Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW), pp. 170–177, 2010.
[4] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[5] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber, Bigtable: a distributed storage system for structured data, ACM Trans. Comput. Syst. (TOCS) 26 (2) (2008) 4.
[6] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Proceedings of the 26th Symposium on Mass Storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.
[7] Apache Software Foundation, Apache HBase. <http://hbase.apache.org/>, accessed 2013a.

[8] Apache Software Foundation, Apache Accumulo. <http://accumulo.apache.org/>, accessed 2013b.
[9] A. Fuchs, Accumulo – Extensions to Google's Bigtable Design, Tech. Rep., National Security Agency, 2012.
[10] M. Hausenblas, J. Nadeau, Apache drill: interactive ad-hoc analysis at scale, Big Data 1 (2) (2013) 100–104.
[11] S. Melnik, A. Gubarev, J.J. Long, G. Romer, S. Shivakumar, M. Tolton, T. Vassilakis, Dremel: interactive analysis of web-scale datasets, Proc. VLDB Endowment 3 (1–2) (2010) 330–339.
[12] A. Thusoo, J.S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, R. Murthy, Hive: a warehousing solution over a map-reduce framework, Proc. VLDB Endowment 2 (2) (2009) 1626–1629.
[13] C. Olston, B. Reed, U. Srivastava, R. Kumar, A. Tomkins, Pig latin: a not-so-foreign language for data processing, in: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM, 2008, pp. 1099–1110.
[14] Cascading Project, Cascading – Platform for Big Data. <http://www.cascading.org/>, accessed 2013.
[15] EsperTech Inc., Esper – Complex Event Processing. <http://esper.codehaus.org/>, accessed 2013.
[16] M. Ali, An introduction to microsoft sql server streaminsight, in: Proceedings of the 1st International Conference and Exhibition on Computing for Geospatial Research & Applications, ACM, New York, NY, USA, 2010.
[17] G. De Francisci Morales, SAMOA: a platform for mining big data streams, in: Proceedings of the 22nd International Conference on the World Wide Web Companion, 2013.
[18] C.R. Kalmanek, I. Ge, S. Lee, C. Lund, D. Pei, J. Seidel, J. Van der Merwe, J. Ates, Darkstar: using exploratory data mining to raise the bar on network reliability and performance, in: Proceedings of the 7th International Workshop on Design of Reliable Communication Networks, IEEE, 2009, pp. 1–10.
[19] L. Golab, T. Johnson, J.S. Seidel, V. Shkapenyuk, Stream warehousing with datadepot, in: Proceedings of the 35th SIGMOD Conference on Management of Data, ACM, 2009.
[20] V. Kumar, H. Andrade, B. Gedik, K.-L. Wu, DEDUCE: at the intersection of MapReduce and stream processing, in: Proceedings of the 13th International Conference on Extending Database Technology, ACM, 2010, pp. 657–662.
[21] S. Efftinge, M. Völter, oAW XText: a framework for textual DSLs, in: Proceedings of Eclipse Modeling Symposium at Eclipse Summit, 2006.
[22] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, M. Zenger, An overview of the Scala programming language, Tech. Rep., EPFL Lausanne, IC/2004/64, 2004.
[23] D.E. Bell, L.J. La Padula, Secure computer system: unified exposition and multics interpretation, Tech. Rep., MITRE Corporation, ESD-TR-75-306 (MTR-2997), 1976.
[24] R. Vanbrabant, Google Guice: Agile Lightweight Dependency Injection Framework, Apress Media LLC, 2008.
[25] L. Tassiulas, A. Ephremides, Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks, IEEE Trans. Autom. Control 37 (12) (1992) 1936–1948. ISSN 0018-9286.
[26] E. Smith, JVM Serializers Project. <https://github.com/eishay/jvm-serializers/wiki>, accessed 2013.
[27] T. Aihkisalo, T. Paaso, A performance comparison of web service object marshalling and unmarshalling solutions, in: Proceedings of the 2011 IEEE World Congress on Services, IEEE, 2011, pp. 122–129.
[28] P. Coetzee, S. Jarvis, CRUCIBLE: towards unified secure on- and off-line analytics at scale, in: Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems, DISCS-2013, ACM, 2013, ISBN 978-1-4503-2506-6, pp. 43–48. <http://doi.acm.org/10.1145/2534645.2534649> .
[29] M. Thompson, D. Farley, M. Barker, P. Gee, A. Stewart, Disruptor: high performance alternative to bounded queues for exchanging data between concurrent threads. <http://disruptor.googlecode.com/files/Disruptor-1.0.pdf>, 2011.
[30] M. Zaharia, M. Chowdhury, M.J. Franklin, S. Shenker, I. Stoica, Spark: cluster computing with working sets, in: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing, pp. 10–17, 2010.